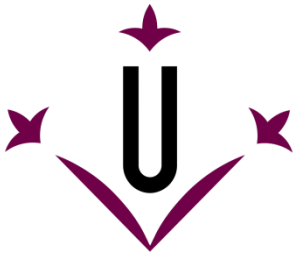
UNIVERSITAT DE LLEIDA
Escola Politècnica Superior



Deduplication of Universitat de Lleida scholarly data

Master's Final Project

Albert Berga Gatius
Tutored by: Roberto García González
01/07/2017

Content

Table of figures

Table of tables

Table of snippets of code

1 Objective

Nowadays, every research institution has a repository where they have a control of the publications done by their researchers. The problem is that in some cases, the researchers themselves are responsible for introducing their papers into the repository and it might cause some data inconsistencies. This issue particularly happens on the GREC[1] (Research management of the Universitat de Lleida) repository of the Universitat de Lleida.

The problem is that, since research publications are usually done by more than one person, sometimes also more than one person introduces the publication information to the GREC repository, which results in duplicated data. Furthermore, most of the times they do not add the same information. Some of them leave blank fields; others mistype some words, etc. In the end, what we have in the GREC repository is duplicated publications difficult to identify because their information does not match.

In this project, we are going to work with a dataset from the GREC repository in order to detect duplicated information about the articles. To do so, we will use data science techniques and algorithms so that we can also explore a bit more this computer-engineering field. Obviously, it will be easy to detect a duplicate when two people introduce the publication in the same way. Therefore, the difficulty of the project will be focused on how to detect duplicates when the information is not exactly the same.

[1] http://webgrec.udl.cat/

2 Motivation

Before starting this project, I had been working on a similar project but mostly developed with Pandas[2] and just a little bit with Spark. Through this project, I want to keep working in Spark to see its possibilities and get a more reliable impression of it. Additionally, using Scala language is another motivation since I have read some opinions saying that it is an enjoyable language. At a glance, just for the fact that it combines Object-Oriented and Functional programming, I think I will enjoy it.

Apart from this, deduplication on scholarly data is something that is being explored by some researchers and can be a way of introducing myself into research and maybe extend the project with a paper. In fact, the deduplication process we will follow is inspired by the article (Zhang, Z., Nuzzolese, A. G., & Gentile, A. L., 2017), which we will explain in the next section.

Finally, another motivation is to deal with some semantic web concepts and technologies. The last part of the project will be focused on uploading the data to an SPARQL repository and processing it from a server side application, which is something I have never done before. Unfortunately, this part might not be done because of the lack of time, and we will have to leave it as future work.

[2] http://pandas.pydata.org/

3 Inspiration

As we have said before, the deduplication process we will apply is based on the one explained in (Zhang, Z., Nuzzolese, A. G., & Gentile, A. L., 2017), whose main purpose is defined as follows:

> *"Given a set of URIs $E = \{e_1, e_2, \ldots, e_m\}$ representing entities of the same type, the goal of deduplication is to: (i) identify sets of duplicate URIs that refer to the same real-world entity. We will call such URIs in each set ' co-referent ' to each other; (ii) determine in each subset one URI to keep (to be called the ' target URI '), while deprecating the others (to be called the ' duplicates ') and consolidating RDF triples from the duplicates into the target URI."*

The authors divide this process into the following parts.

3.1 Blocking

In order to detect duplicated entities, they have to compare them pairwise. However, comparing the whole dataset in that way has a quadratic cost, so the idea is to create subsets and compare only entities of the same subset. The blocking phase is focused on creating these subsets and they propose two different solutions.

The first one consists on ordering the items lexicographically and produce all pairs for e_i with all e_j in a context window of size *n.* By this way, all items will be compared with their *n* lower and upper neighbours. As we can see in the Figure 1, this method reduces the amount of comparisons. This figure shows which comparisons would be done (green cells) applying this method to a dataset of 10 elements with a window size of 4. Whereas white cells are comparisons that have been avoided with the use of this method.

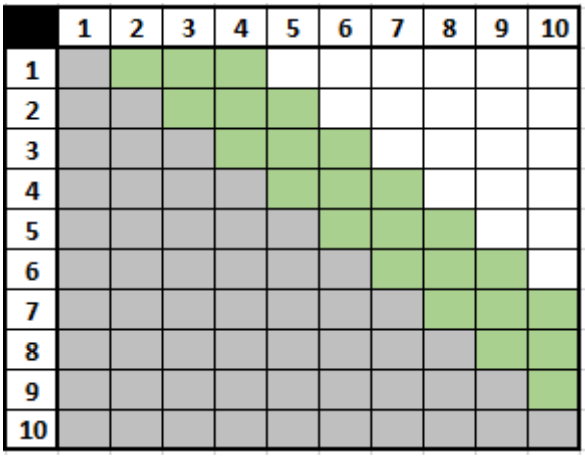


Figure 1 Blocking with context window

The second technique is content based and they create directly candidate pairs rather than blocks. The idea is to create a candidate pair for each pair of items that share at least a common value in any of their properties. With this method, they will only compare pairs of elements which similar in any way.

3.2 Classification

Once they have the candidate pairs, they have to compare them to determine if both items are referring the same one or not. Since their dataset contains linked data, different items may contain different properties referring to similar information. In order to compare as much information as possible, they first gather all properties of each entity and classify them into different features, according to what the properties refers to. At this point, they have a set of values for each feature of the items. Therefore, the comparison of two items becomes now a comparison of different pairs of sets. To deal with these comparisons, they use functions based on the Dice co-efficient[3], which evaluate how the information that both items have in common describe them, and functions based on the Coverage, which evaluate the maximum degree to which the common part of the two can describe either of them. This produces, for each compared pair, a vector of dimension equal to the number of features, with the coefficients obtained throughout the previously mentioned functions. After that, they experiment with different binary classification models to see

[3] https://en.wikipedia.org/wiki/S%C3%B8rensen%E2%80%93Dice_coefficient

which one works better with the task of determining whether two items are referring the same or not.

3.3 URI harmonisation

Once they know which URIs are co-referent, they have to decide which ones to keep as target and which ones as co-referent. This decision is made depending on the indegree and outdegree of the entity, which counts the number of incoming and outcoming ties of each node. Additionally, in order to keep the principal URI as complete as possible, they associate with it the knowledge available through the other co-referent URIs. After that, the co-referent URIs will be associated with the principal one through a new property.

4 Approach

In this project, we will try to detect and mark/remove the duplicates in a dataset of scholarly data from GREC. Even though the dataset is not very large, the idea is to use Spark, which is often the best option for data science solutions but it is usually used to process large volumes of data that cannot be loaded in memory. However, using Spark we make our solution scalable and ready to work with larger volumes of data.

The way we are going to do the deduplication process will be based on the procedure explained in the article seen in the previous section. We are going to use string metrics to detect fields that are nearly equal in two items and, when most of the fields are almost the same, we will consider them as referring the same article. Again, in the same way as in the solution proposed in (Zhang, Z., Nuzzolese, A. G., & Gentile, A. L., 2017), we will not remove the duplicate articles but relate them with the most informative one.

As the previous process requires comparing articles pairwise, it would be insane to do it over the whole dataset and we might not be able to do it due to the computational cost it would take. Therefore, we should first group them in a way that articles that are more likely to be co-referent are classified into the same group, as also done in (Zhang, Z., Nuzzolese, A. G., & Gentile, A. L., 2017). By this way, we will only compare the articles after grouping them into smaller sets. Nonetheless, maybe we will have to repeat the process some more times varying a bit the groups to compare the articles from different groups too. It all depends on the way we group them.

After that, we will have each duplicated article linked with another one. Therefore, for each group of duplicated articles, we will have them related forming a graph. However, it has no sense to have them related in this way, and we will need to have a principal row and all others related to it. So we will set up a final procedure to do this transformation.

5 Success measures

In order to measure the accuracy of our solution, it seems obvious that we have to count how many duplicates have been correctly detected and how many have not. However, this idea is well defined by "Precision and recall"[4], which is a way of measuring the success on any progress on which the objective is to detect a set of elements matching a condition.

5.1 Precision and Recall

Figure 2 helps a lot to understand Precision and Recall. It shows how different elements can be classified whenever the intention is to find elements matching a condition. Firstly, elements can match the condition (left side) or not (right side). Then comes the solution, after which elements will be again classified as matching or not the condition, but from the point of view of the solution (circle). As we can see in the image, at this point, we can classify the elements resulted from the solution into four different types.

- False negatives: elements badly marked as not matching the condition.
- True negatives: elements correctly marked as not matching the condition.
- True positives: elements correctly marked as matching the condition.
- False positives: elements badly marked as matching the condition.

Precision and recall are defined using the previous naming as follows:

$$Precision = \frac{true\ positives}{true\ positives + false\ positives}$$

$$Recall = \frac{true\ positives}{true\ positives + false\ negatives}$$

Precision measures the correctness of the selected elements while recall measures the portion of all elements matching the condition that have been selected. Both are very important but having two values to measure the

[4] https://en.m.wikipedia.org/wiki/Precision_and_recall

accuracy of a solution is a bit awkward. We need to combine them into a single value, which is calculated through the F-measure.

$$F - measure = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

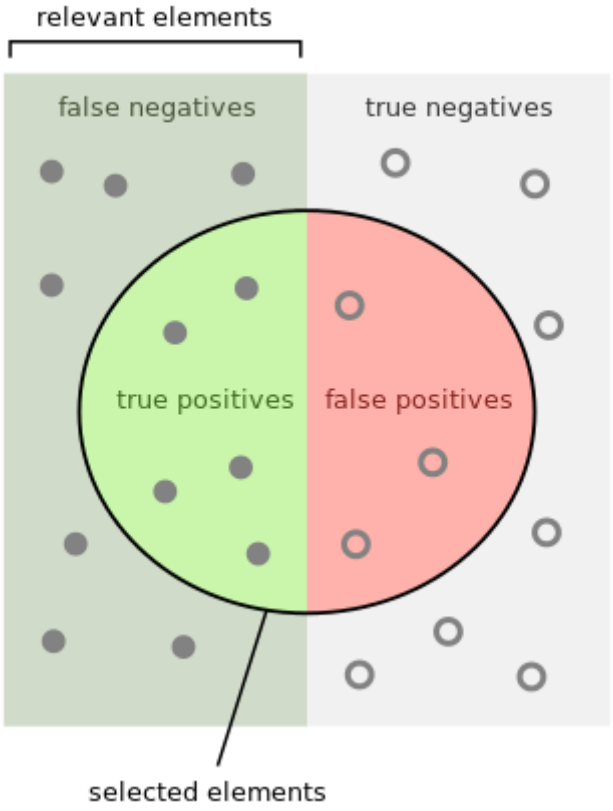


Figure 2 Precision and Recall elements classification

5.2 Enforcement

To apply the previous measures we need certain about which rows are duplicated and which not. However, we do not have this information, so we will have to deduplicate the dataset manually, or at least a subset of it, so that we can compare it with the results of the solution. Eventually, we will be able to calculate the F-measure so that we obtain a value indicating the accuracy of our solution.

6 Budget

After searching for some data scientist job offers on the internet[5] we have reached the conclusion that the salary of a junior data scientist is around 20.000€ per year for a full-time job. Nonetheless, our project will not be a full-time project and neither will we work for a complete year. Thus, the easiest way to calculate the necessary budget to get ahead the project is calculating the price per hour of a junior data scientist, which can be calculated from the annual salary.

$$\text{Annual worked hours} = 8 \text{ hours a day} \cdot 5 \text{ days a week} \cdot 52 \text{ weeks a year} = 2080 \text{ hours a year}$$

$$\text{Price per hour} = \frac{20.000 \text{ € / year}}{2080 \text{ hours / year}} \simeq 10\text{€/hour}$$

As a result of the previous calculations, we obtain that the price of an hour of work of a junior data scientist is more or less 10€.

On the other hand, in this project there will be a person (myself) working an average of 4 hours a day during 4 months. Therefore, we can do some more calculations to obtain the total amount of money necessary to develop the project.

$$Hours\ per\ month = 4\ hours\ a\ day \cdot 5\ days\ a\ week \cdot 4\ weeks\ per\ month = 80\ hours\ /\ month$$

$$Monthly\ salary = 80\ hours\ per\ month \cdot 10\text{€}/hour = 800\text{€}/month$$

$$\boldsymbol{Project\ total\ cost = 800\text{€}/month \cdot 4\ months = 3200\text{€}}$$

[5] https://www.glassdoor.com/Salaries/spain-junior-data-scientist-salary-SRCH_IL.0,5_IN219_KO6,27.htm

7 Planning

Before starting the development of the project, we have defined some iterations in order to divide the implementation into progressive steps. By this way, at the end of each iteration we will be able to document the current state of the solution as well as the problems encountered during the development of the iteration.

On the other hand, although we define all the iterations now, these could suffer some changes during the course of the project caused by different troubles in any parts of the solution or not having enough time to do all work. Therefore, what we define now will be a first approach to the planning of the project. However, while developing the solution, we will set up the objectives of each iteration before starting each of them based on the result of the previous one.

Next, we briefly define these iterations.

1. **Iteration 1:** The first iteration will be focused on cleaning the dataset and dividing it into two different subsets: one for the author names and another one for articles' information.
2. **Iteration 2:** The aim of the second iteration will be to define and implement the deduplication process. In this implementation, we will use basic functions to group articles and comparing them. In the same way, to select which item to keep as principal, we will just keep the first one. The definition of different functions for these parts will be done in the next iteration.
3. **Iteration 3:** In this iteration, we will define and implement different functions for the grouping process and for determining if two articles are near duplicates. Moreover, we will define some function to determine which of the duplicated articles keep as principal.
4. **Iteration 4:** The objective of this iteration will be to compare the results obtained applying different combinations of the functions defined in the previous iteration. In order to measure the correctness of our results, we

will manually deduplicate a subset of our dataset and apply the success measures defined previously.

After defining the different iterations, we have to do a temporal planning to have some time reference about when we should finish each iteration and continue with the next one. Nonetheless, as the iterations defined previously may suffer some changes during the course of the project, their temporization may also change. In fact, also the end of the project could suffer some changes if the development of it does not go as expected.

The following GANTT diagram shows a first approach of ideal temporization of the project. Note that we also include the temporization of the documentation part of the project and the lecture preparation.

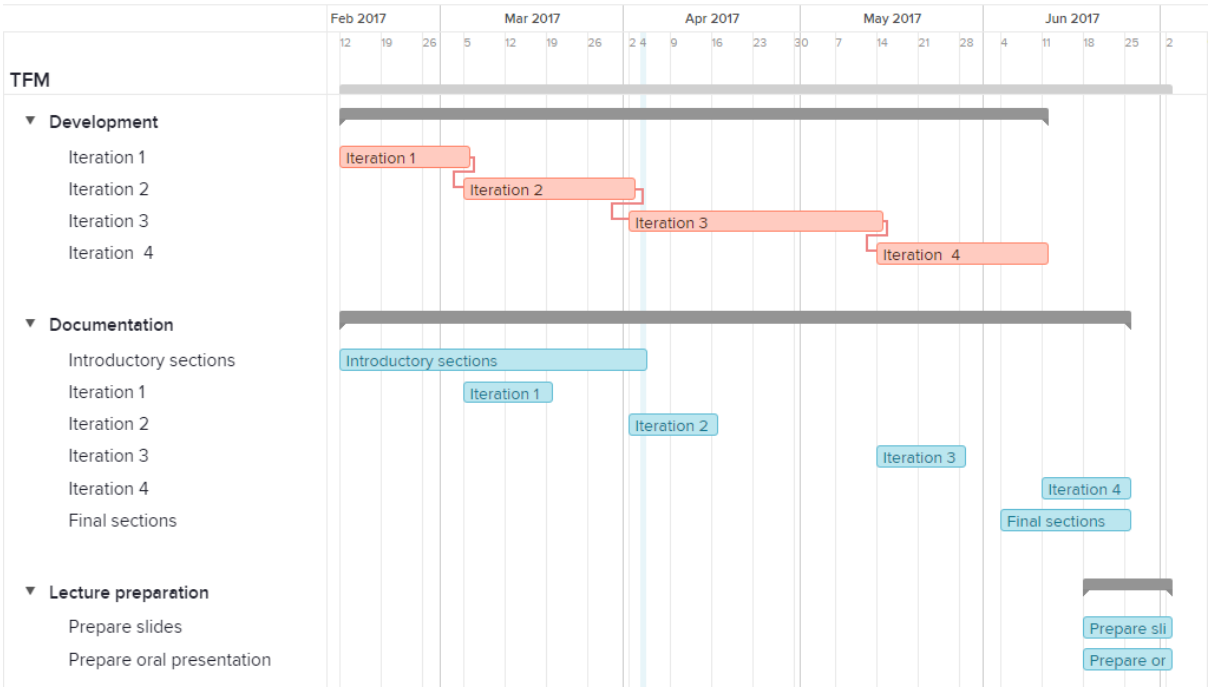


Figure 3 Planning GANTT diagram

8 Technologies used

8.1 Apache Spark

Apache Spark is an open-source engine developed specifically for handling large-scale data processing and analytics. It is designed to accelerate analytics on Hadoop while providing a complete suite of complementary tools. It was released in May 2014, and since then it has become very popular both with developers and in enterprises. The reason for this popularity is its increase in speed and efficiency over Hadoop MapReduce, as both are used for similar tasks.

Since Apache Spark 1.6, and consolidated with Apache Spark 2.0 release, it has been added the Datasets API to the already existent DataFrames and RDDs. It is the last attempt from Apache to make their APIs easy to use, intuitive and expressive, in the same way, that they had introduced DataFrames API in the 1.3 version. Next, we are going to do a brief introduction to each of these APIs to clarify what they are focused on.

8.1.1 RDDs

RDD was the principal user-facing API in Spark since its inception. At the core, an RDD is an immutable distributed collection of elements of your data, partitioned across nodes in your cluster that can be operated in parallel with a low-level API that offers transformations and actions.

This is the most cumbersome way to work in Spark because data is unstructured. Unfortunately, in some cases, there is no other choice because the data does not follow a structure or you just do not care about imposing a schema. Nonetheless, you must take into account that DataFrames and Datasets have some optimization and performance benefits that are lost using RDDs.

8.1.2 DataFrames

Like an RDD, a DataFrame is an immutable distributed collection of data. Unlike an RDD, data is organized into named columns, like a table in a relational database. Designed to make large data sets processing even easier, DataFrame allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction; it provides a domain specific language API to manipulate your distributed data.

This is the only option to work with structured data when the programming language is Python or R. Otherwise, if we are working with Scala or Java we will use Datasets API as it unifies both Datasets and DataFrames APIs.

8.1.3 Datasets

Introduced in Apache Spark 1.6 as an experimental part and consolidated on Spark 2.0, Datasets takes on two distinct APIs characteristics: a strongly typed API and an untyped API, as shown in the Figure 4 below. Conceptually, consider DataFrame as an alias for a collection of generic objects Dataset[Row], where a Row is a generic untyped JVM object. Dataset, by contrast, is a collection of strongly typed JVM objects, dictated by a case class you define in Scala or a class in Java.

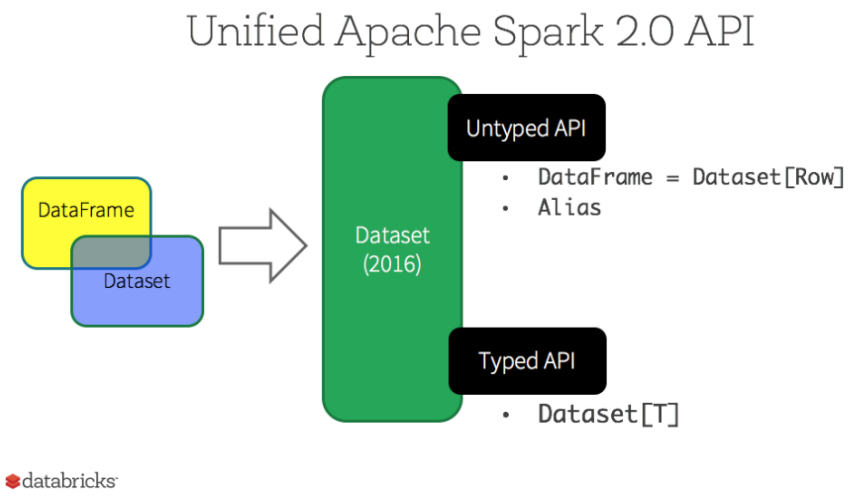


Figure 4 Apache Spark 2.0 API: Structure

So DataFrames and Datasets have been unified in Apache Spark 2.0, which means we can benefit from both APIs at the same time. Furthermore, Dataset APIs provides much more benefits, which we are going to explain next:

- As it is a strongly typed API, most errors will be detected at compile time. This can save a lot of time when working with high volume of data.
- A strongly-typed Dataset[T] can be easily displayed or processed by high-level methods.
- Availability of RDD's functions with structured data.

Along with all the above benefits, you cannot overlook the space efficiency and performance gains in using DataFrames and Dataset APIs for two reasons. First, because DataFrame and Dataset APIs are built on top of the Spark SQL engine, which uses Catalyst to generate an optimized logical and physical query plan. Second, since Spark as a compiler understands your Dataset type JVM object, encoders can efficiently serialize and deserialize JVM objects as well as generate compact bytecode that can execute at superior speeds.

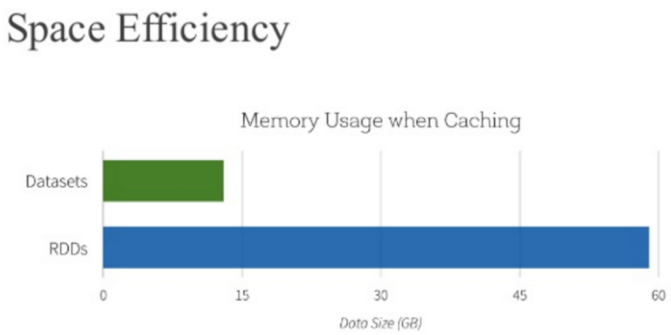


Figure 5 Apache Spark APIs: Space efficiency

8.1.4 Our choice

It seems clear that whenever possible, the most attractive option is using Datasets API. As our data comes from CSV files, we can structure it into a

typed Dataset[T]. However, we will have to work in Scala if we want to work with an interactive and reactive environment like IPython-style notebooks[6].

8.2 Databricks

Databricks was founded by the team that created Apache Spark with the intention of making it easier to extract value from Big Data, as they believe that it is still largely untapped. In this way, they provide a web-based platform for working with Spark with automated cluster management and IPython-style notebooks. Apart from the Databricks platform, the company is co-organizing massive open online courses[7] about Spark and runs the largest conference about Spark - Spark Summit.

We could implement our entire project within the Databricks infrastructure. However, we are not going to use it for the final solution but only for some intermediate steps. It is a very good option for trying sparking functions or different implementations and seeing the results immediately. Nonetheless, the free cluster they provide is limited to 6 GB of RAM, which is not a lot for a data science project. For this reason, when we want to execute the whole solution with all the data, we are going to do it locally.

8.3 Scala language

Scala is an acronym for “Scalable Language”, which means that it is very scalable and prove of it is that it is used by companies like Twitter, LinkedIn or Intel for their large mission critical systems. The generated code is on a par with Java’s, and its accurate typing means that many problems are caught at compile-time rather than after deployment. At the root, the language scalability is the result of a careful integration of object-oriented and functional language concepts.

[6] https://en.wikipedia.org/wiki/Notebook_interface

[7] https://en.wikipedia.org/wiki/Massive_open_online_course

8.3.1 Object oriented or functional language?

Scala is a purebred object-oriented language. Conceptually, every value is an object, and every operation is a method call. The language supports advanced component architectures through classes and traits. However, it is also a full-blown functional language. It has everything you would expect, including first-class functions, a library with efficient immutable data structures, and a general preference of immutability over mutation.

8.3.2 Java interop

Scala runs on the JVM. Java and Scala classes can be freely mixed, no matter whether they reside in different projects or in the same one. They can even mutually refer to each other; the Scala compiler contains a subset of a Java compiler to make sense of such recursive dependencies.

Java libraries, frameworks and tools are all available. Build tools like ant or maven, IDEs like Eclipse, IntelliJ, or NetBeans, frameworks like Spring or Hibernate all work seamlessly with Scala. Scala runs on all common JVMs and also on Android.

9 Development

9.1 Iteration 1

9.1.1 Objectives

The first iteration of the project consists on analysing the data we are going to use for the project and prepare it to start the deduplication process. Firstly, we are going to look at each field of the dataset analysing its possible values and the coherence of these values. After that, we will apply some data cleaning processes to give more consistency to the data. Additionally, we must think about which format the data will have during the deduplication process. Since it is not enough to just remove the duplicated rows because we would be losing important information, we will relate by some way the different rows that refers to the same article. Therefore, the structure of the data must support this relationship and is in this iteration when we must define it.

9.1.2 Solution

First, we are going to describe the different fields of the dataset so that we can get a clear idea of what information we have about each article. We will explain the fields into different groups according to what they refer to.

The first group of fields refers to a person who is linked with the article information indicating that he is one of the authors of it. In some cases, this person will be who has introduced the article information to the GREC repository. In other occasions, this person will not have directly introduced it to the GREC repository but only has linked the article to his profile. In both cases, the format of the article information will be the same.

- NIF: NIF of the person linked with this article.
- name: name of the person linked with this article

Next, we have some information about the article itself. Note that there is a ‘code’ field, which is assigned by the GREC repository when the article is

submitted for the first time. However, it is not unique over the dataset because for each author linked with the article information there will be a different row with the same code.

- code: code identifying the article in the GREC repository.
- publicationYear: the year when the article was published.
- title: title of the article.
- nAuthors: number of authors of the article.
- authors: list of authors of the article.

The next group of fields are about the journal in which the article was published.

- volume: the volume of the journal in which the article has been published.
- numJournal: the volume number in which the article has been published.
- iniPage: page where the article starts inside the journal.
- endPage: page where the article ends inside the journal.
- DOI: acronym for Digital Object Identifier.
- journalType: GREC acronym of the type of journal.
- journalTypeDesc: Description of the type of journal.
- issn: acronym for International Standard Serial Number.
- journalCode: code identifying the journal in the GREC repository
- journalDesc: description of the journal.
- impactFactor: impact factor of the journal[8].
- classification: classification of the journal according to its impact factor.

Although all fields seems to be very clear, we have detected some fields with strange values or empty values. In order to avoid future problems caused by these wrong values, we have done the following cleaning operations:

- Fill up empty values with “0” in fields whose value are always integer so that we can convert the field type to integer in case we need to do integer operations with any of these fields.

[8] https://en.wikipedia.org/wiki/Impact_factor

- Match four consecutive digits in “publicationYear” or set value 0 if no matching found. We have detected some cases in which this field had also the month.
- Replace “,” by “.” in “impactFactor” so that we can convert the field type to Double.

Even though we have tried to convert to Integer type the fields where has no sense having any other value, there are some fields in which we have preferred to keep the strange values and do not change the field type. By this way, we do not lose information that might be useful in some point in the future. For example, “iniPage” and “endPage” sometimes had values like “085606-5”. Maybe the first part of the value corresponds to the “numJournal” and it helps us to determine that two rows refers to the same article.

After that, we have had to decide how we would indicate that some articles are referring the same one. As we do not want to lose information, removing the duplicated rows was not an option. Therefore, we have added a new column to the data, called “sameAs”, to specify that a given article is referring the same as another one. The task of detecting duplicates is not part of this iteration but there are some rows in which the only thing that changes is the information about the person with whom this article is linked. To avoid this problem we are going to separate the dataset into two new ones; the first one containing the information about the authors linked with each article and the second one containing the information about the articles. After that, we can remove the duplicated rows from the articles dataset, which belonged to each of the authors related with an article. The following piece of code shows how we do that:

```
val journalAuthors = articles.select("nif", "name", "code")
                                .as[JournalAuthor]
val journalArticles = articles.drop("nif").drop("name")
                                .dropDuplicates().as[JournalArticle]
```

Code 1 Dataset columns' division

The ‘as’ function stores the dataset as rows of the given scala case class, which we have previously created.

9.1.3 Later Improvements

The previously explained data cleaning processes were implemented following data science theories saying that the first step for a data science project consists on leaving the data as clean as possible, which includes transforming each column type into the corresponding one (e.g. Integer, Double, …) and filling null or empty values with some neutral value. Nonetheless, some of these processes have been removed from the final solution because of some changes done while implementing the function to compare two articles in the Iteration 4.

On the one hand, we have left the null values as they were and all columns types as strings for two reasons. Firstly, we do not need numeric columns in any numeric type because we only use these values to compare different articles and all comparisons are string-based. Secondly, in our comparative function, we do not take into account a field when it contains a null value in any of the two articles being compared. So filling null values is only a trouble that makes the comparative function look messier.

On the other hand, the type of the field “impactFactor”, which had been set as Double, has also been left as String. In fact, this field is not even used in the comparative function so it does not matter which type it has. However, since we have not filled null values, our implementation failed replacing commas by dots so we have decided to avoid this transformation.

9.1.4 Conclusion

In this first iteration we have seen the difficulties of cleaning the data and not losing useful information since sometimes it is incompatible and we have to decide whether the information is important enough to not cleaning it or we prefer losing information but getting the data in the right format.

Additionally, it is very important to know the context of the data, how this data is generated and obtained. Otherwise, we would hardly know things like that when two rows have the same “code” means that it is surely the same article but with two people linked to it.

9.2 Iteration 2

9.2.1 Objectives

The second iteration of the project is focused on defining and implement the whole structure of the deduplication process. It means that after this iteration we should be able to run our deduplication process over a dataset and obtain a deduplicated version of it. However, the resulting dataset will be mostly inaccurate because the functions in charge of the important deduplication procedures will be implemented as stubs. The implementation of these functions will be done in the next iteration.

9.2.2 Solution

As we have detailed in the section Inspiration, we are going to follow more or less the same structure showed in the article. Therefore, our implementation will have mainly four parts: sorting, blocking and Cartesian product, classification or deduplication and principal row selection. We are going to explain each of them in detail next.

9.2.2.1 Sorting

As we are not going to compare all rows in the dataset pairwise but only the rows in the same block, we have to ensure that rows of the same block are as likely as possible of being duplicates. To do it, we are going to sort the rows in order to leave as close as possible the similar rows. In this iteration, we have just ordered the rows by title, as we can see in the following code:

```
val sortedByTitle = articles.orderBy("title")
```

Code 2 Sort articles by title

9.2.2.2 Blocking and Cartesian product

With the rows sorted, we have to group them into different blocks and apply the Cartesian product to each block with itself. By this way, we will obtain a dataset with each row containing two articles, which we can compare easily in the next step.

For this first implementation, we have just grouped the rows ten by ten. Of course, this blocking strategy has a major problem, which is that rows are only compared with other rows in the same block, but what we want in this iteration is the implementation of the whole structure so it is fine for now. The following piece of code shows the function in charge of the blocking and Cartesian product:

```
def divideAndCartesian(data: Dataset[IndexArticle], divisionSize: Int):
                                                Dataset[List[IndexArticle]] = {
      data.groupByKey((a: IndexArticle) => a.index / divisionSize)
          .flatMapGroups((l: Long, x: Iterator[IndexArticle]) =>
                          x.toList.combinations(2))
}
```

Code 3 Blocking and Cartesian product

Before calling this function, we zip the articles dataset so that we can use the index to create blocks of N rows. As we can see, this function receives two parameters: the dataset to block and apply Cartesian product and the size of the block. What we do is grouping the dataset by groups of the specified size and, for each group, we use the Scala list's function "combinations", which returns lists with the different possible combinations on N elements from the list (where N is a parameter of the function).

9.2.2.3 Deduplication

With the dataset resulting from the previous step, it is so easy to compare the rows. As each row contains two articles, we just have to filter the dataset with a filter function which compares both articles of each row and determines if they are referring the same article.

In this case the comparative function simply returns true if more than a half of the fields are the same in both articles. But the interesting part of this function is that we can implement it inside the Scala case class we use to encapsulate rows:

```
case class JournalArticle(code: Integer, publicationYear: Integer, title: String,
                          nAuthors: Integer, authors: String, volume: Integer,
                          numJournal: String, iniPage: String, endPage: String,
                          DOI: String, journalType: String, journalTypeDesc: String,
                          issn: String, journalCode: Integer, journalDesc: String,
                          isiCode: Integer, impactFactor: Double,
                          classification: String, sameAs: Integer){
      def isNearDuplicate(article: JournalArticle): Boolean = {
             val numEqualFields =
                    this.productIterator.zip(article.productIterator)
                           .count((articles) => articles._1 == articles._2)

             numEqualFields >= this.productArity / 2
 }
}
```

Code 4 Article's case class

9.2.2.4 Principal row selection

The aim of this phase is to select the row that best explains the duplicated article in order to keep it as principal and make all other duplicated articles refer this one. Nonetheless, we have another problem before we can select the principal row, which is that we have a list of rows referring other rows, but what we need is a list of groups of articles that are referring the same article. In other words, we have different graphs (one for each deduplicated article) and we need to get a list of nodes for each non-connected graphs.

The Spark's GraphX library[9] helps a lot when working with graphs in Spark and it also has a function that does exactly what we need. Specifically, given a list of nodes and edges, the function "connectedComponents" separates the different non-connected graphs by identifying each node with the smallest node of the connected graph it belongs to.

For a first implementation, it is ok to make all duplicated articles to refer the one with smallest code, so we just need to join the result of the

[9] https://spark.apache.org/graphx/

"connectedComponents" function with the whole articles dataset. We can see how we do it in the following piece of code:

```
val cc = graph.connectedComponents()
val deduplicated = articles.drop("sameAs")
             .join(cc.vertices.toDF("code", "sameAs"), Seq("code"),
                   "left_outer")
             .as[JournalArticle]
```

Code 5 Fill "sameAs" field from GraphX's "connectedComponents"

9.2.3 Problems

Even though implementing this iteration had not been much difficult initially, we had implemented the Blocking and Cartesian product part through a recursive function that was causing the execution to waste much memory and preventing it to finish unless the amount of data to deduplicate was only a few rows. In the following piece of code, we can see this first implementation.

```
def divideAndCartesian(data: Dataset[IndexArticle], fromIndex: Long,
                       divisionSize: Int): Dataset[CartessianIndexArticles] = {
val division = data.filter(x => x.index >= fromIndex
                                  && x.index < fromIndex + divisionSize)
if(division.count() == 0)
       Seq.empty[(JournalArticle, Long, JournalArticle, Long)]
       .toDF("article1", "index1", "article2", "index2").as[CartessianIndexArticles]
else
       division.crossJoin(division)
       .toDF("article1", "index1", "article2", "index2")
       .as[CartessianIndexArticles]
       .union(divideAndCartesian(data, fromIndex + divisionSize, divisionSize))
}
```

Code 6 Division and Cartesian wrong implementation

In this implementation, we are filtering the chunk of data we want to process, then we apply the Cartesian product of the chunk with itself and we recursively join this result with the result of applying the function to the next chunk of data.

We discovered that this part of the code was preventing the process to finish after looking at the Spark's UI, where we can see the details of each job. There we saw that the "divideAndCartesian" function was creating thousands of stages (a Spark's job is divided into stages).

9.2.4 Conclusion

In this iteration we have seen how important can be to choose the most suitable alternative whenever we have to implement a feature. Even though the program works and does what it has to, it is important to check the solution and think about a better implementation in order to avoid future problems. In our case, we have been lucky that the execution time was so long that the program could not finish. Otherwise, we may have not detected the problem unless we executed the program with a larger dataset. To avoid this kind of problems, we always have to evaluate if the execution can be faster because a little difference in the execution time can grow exponentially when the data to process increases.

9.3 Iteration 3

9.3.1 Objectives

According to the previous iteration, the aim of this one should be to implement the key parts of the deduplication process, which are currently implemented as stubs. Therefore, there were four key points to implement:

1. Sort the dataset so that neighbours are as likely as possible to be referring the same article.
2. Blocking the articles and apply Cartesian product of blocks in a way that each article is joined with its top and bottom N neighbours.
3. Compare each pair of articles and determine whether they are similar enough to be marked as duplicate or not.
4. Calculate the sets of duplicates of the same articles (as they will be compared pairwise) and decide which row to keep as principal, so that all other rows reference that one.

However, before starting this iteration, we have done some research about the different alternatives to avoid having to compare pairwise all rows. We have found that it is not possible to implement such a function just by sorting different string based rows so that similar strings are sorted more or less in the same position. For instance, two items' title might start different due to some typographic error, and thus sorted apart, while they might correspond to the same article. Therefore, in order to overcome this problem, we have looked for other alternatives and we have found the Locality-sensitive Hashing algorithm (LSH), which automatically detects blocks of similar items. Therefore, LSH will replace the first step and the blocking part of the second one. Then, to reach the third step we will just have to apply Cartesian product over the articles of each block of similar items detected by LSH.

9.3.2 Solution

9.3.2.1 Locality-sensitive Hashing

The general idea of LSH is to classify the elements of a set into different blocks in a way so that similar elements are more likely to be classified into the same block than dissimilar items are.

However, it is not possible to apply LSH directly to a set of strings. It can only be applied to a set of different signatures representing the elements of the original set. Moreover, depending on how these signatures are calculated, a different distance measure will be used to determine how similar elements are. Some of these distances are Jaccard similarity, Hamming distance, Cosine distance or Euclidean distance.

In our case, we want to classify into the same block those articles whose titles are similar and we have found that Jaccard similarity of k-shingles of the titles could be a good approximation. We will cover what *k*-shingles are as well as all other steps we have to perform in order to apply LSH to the titles of our articles in the following sections.

Shingling of strings

Shingling is an effective way to represent strings as sets. The idea is to extract the set of substrings of length *k* (*k*-shingles) that appear within the main one. Then, we may associate with each article the set of *k*-shingles that appear one or more times within the article's title. At this point, we have each article represented by a set of *k*-shingles and we could compare different articles by any sets distance, including Jaccard similarity. Nonetheless, we do not want to compare all articles pairwise but apply LSH to obtain blocks containing articles with Jaccard similarity higher than a given *threshold*.

Minhashing

The idea of minhashing is quite simple; it consists on applying a hash function to all elements of a set and get the lowest value. However, in our case it is not enough applying only one hash function because we need a set of signatures for each article in order to apply LSH. So we will apply many hash functions to each set so that the result will be a matrix like the next one:

	S1	S2	S3
h1	$h1_{min}(S1)$	$h1_{min}(S2)$	$h1_{min}(S3)$
h2	$h2_{min}(S1)$	$h2_{min}(S2)$	$h2_{min}(S3)$
h3	$h3_{min}(S1)$	$h3_{min}(S2)$	$h3_{min}(S3)$

Table 1 Minhashing matrix representation

The interesting part here is that the probability that the minhash of two different sets produces the same value for a given hash function equals to the Jaccard similarity of these two sets. We will explain why this happens next.

Minhashing and Jaccard Similarity

The Jaccard similarity coefficient is a commonly used indicator of the similarity between two sets. For two sets *A* and *B*, the Jaccard similarity is calculated as:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

The goal of MinHash is to estimate J(A,B) quickly, without explicitly computing the intersection and union. As explained quite well in the Wikipedia entrance for MinHash[10]:

> "Let *h* be a hash function that maps the members of *A* and *B* to distinct integers, and for any set *S* define $h_{min}(S)$ to be the minimal member of *S* with respect to *h*—that is, the member *x* of S with the minimum value of $h(x)$. Now, if we apply h_{min} to both *A* and *B*, we will get the same value exactly when the element of the union $A \cup B$ with minimum hash value lies in the intersection $A \cap B$. As you may have noted the probability of this to happen is equal to the Jaccard similarity."

LSH for minhash signatures

One general approach to LSH is to "hash" items several times, in such a way that similar items are more likely to be hashed to the same bucket than dissimilar items are. We then consider any pair hashed to the same bucket for any of the hashings to be a *candidate pair.*

If we have minhash signatures for the items, an effective way to choose the hashings is to divide the signature matrix into *b* bands consisting of *r* rows each. For each band, a hash function takes vectors of *r* integers (the portion of one column within that band) and hashes them to some large number of buckets. We can use the same hash function for all the bands, but we use a separate bucket array for each band, so columns with the same vector in different bands will not hash to the same bucket.

[10] https://en.wikipedia.org/wiki/MinHash

Additionally, it is possible to represent the probability that two strings become a *candidate pair* and it results in the following S-curve.

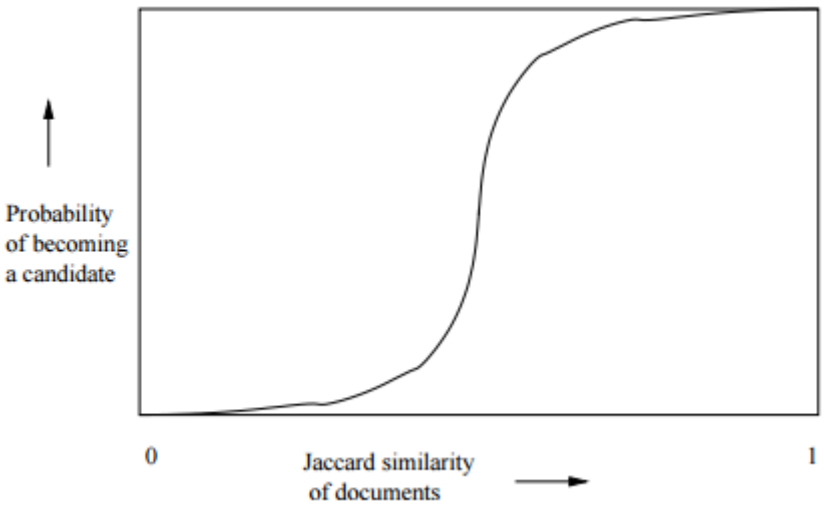


Figure 6 LSH candidate pair detection probability

As you may have noted, the *threshold*, that is, the value of similarity *s* at which the rise becomes steepest, is a function of *b* and *r*. An approximation to the *threshold* is $(1/b)^{1/r}$. So more rows per band means a higher *threshold*, while less rows per band means a smaller *threshold.*

In order to summarize the whole process, the following schema gives a general vision of the different steps to apply LSH to a string-based document.

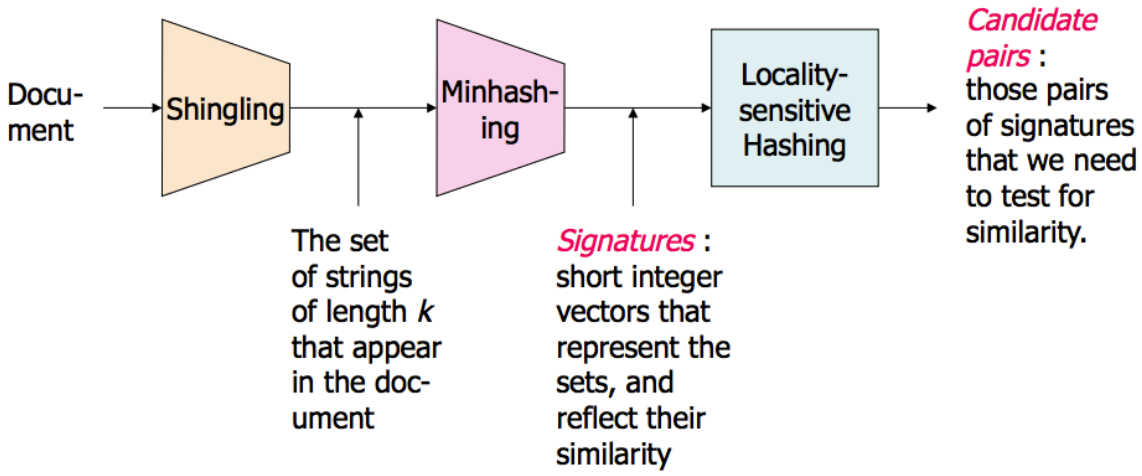


Figure 7 LSH schema for string-based document

9.3.2.2 LSH implementation

In the project, we have implemented a version of the LSH explained before in order to apply it to the title of the articles and obtain groups containing those which have a similar title. We have implemented it in Spark as it will be helpful when we have a large amount of data. Moreover, we have tried to keep the implementation as general as possible so that it can be used for any other data. Given an RDD[(T, String)] where the first element of the tuple is the identifier of the element and the second element is the string from which we will obtain the *k*-shingles, our implementation will return an RDD[List[T]] where each List represents a bucket of the LSH and contains the identifiers of the elements which have been mapped to the same bucket.

Firstly, we calculate the *k*-shingles and the minhash for a given amount of different hash functions. It can be difficult to imagine how to implement a large number of hash functions. However, there are some techniques to "cheaply" calculate different hash values. In our case, we have based on a Stackoverflow discussion[11] about this topic. The idea of a hash function is quite simple; it is a function that produces the same value for the same input and different values for different inputs. Additionally, hash functions for minhashing should produce disperse results so that the lowest hash value not always comes from the same input value. With all that in mind, the first hash value can be obtained through a normal hash function. Then, all other hash values will be obtained from the first hash value combining *bitwise rotation* and an *XOR operation* with a random number. The only thing we have to take into account is that if the random number changes, we obtain a new hash function. So in order to persist the hash functions we have to store the random numbers.

11 http://stackoverflow.com/questions/19701052/how-many-hash-functions-are-required-in-a-minhash-algorithm

Here we can see the implementation:

```
val signatures: RDD[((Int, T), (Int, Int))] =
 data.flatMap({ case (id, text: String) => {
   val shingleList = text.sliding(shingleSize).toList.distinct
   randInts.map({ case (h, i) =>
            ((i % numBands, id),
            (i, shingleList.map(_.hashCode().<<(5) ^ h).min))})
}
```

Code 7 Shingling and obtaining signatures matrix

For each title, the first thing we do is obtaining the *k*-shingles of a string. Scala provides a function to obtain them, which is called “sliding”. Then, we use “randInts”, which is a list of random integers with indexes, to simulate different hash functions (in the way explained before) and apply them to each group of *k*-shingles to obtain the minhash matrix. Note that the result of this process is an RDD[((Int, T), (Int, Int))]. These values correspond to ((#band, article_id), (#hash, minhash_value)). By this way, we leave everything ready to the LSH process, in which all we need to do is grouping the RDD by key and apply a hash function over the values associated to each key. Then, all values which hashes to the same value will be candidate pairs.

9.3.3 Conclusion

First, we have to mention the change of plans we have had to make due to the impossibility of implementing a solution to sort the articles in the way we had thought. However, since our main purpose with this project is to learn and get experience with data science projects, it is good for us to have some troubles so that we see the problems these kind of projects can have. In our case, we have been lucky to find a workaround to the problem of sorting the articles. Otherwise, we might have been forced to give up the project. So from this problem we learn that we have always to search if a solution is affordable before saying we are going to do that. Because in this case it would have been just a university project, but if it were a project for a client, we could have been in serious trouble.

On the other hand, even though in this iteration we wanted to implement both the LSH process and the comparison of the candidate pairs resulting from LSH, we have eventually decided to leave the comparison for the next iteration. This is because, firstly, the load of work of understanding and implementing LSH was big enough to cover an iteration and, secondly, because we have thought that it will be interesting to analyse some of the candidate pairs obtained by LSH in order to see which type of comparison fits better depending on the differences between them.

9.4 Iteration 4

9.4.1 Objectives

This iteration has arisen from the division of the previous one into two different parts; the LSH algorithm and the comparison of candidate pairs detected by it. At this point, we have our articles classified into different groups in a way that articles of the same group have a high Jaccard similarity between *k*-shingles extracted from their titles. So the idea now is to compare pairwise articles from the same group taking into account all their fields and decide whether they refer the same article or not.

We will compare fields from both articles with different comparison functions depending on the type of field. For example, for the field containing the title of the article we will use edit distance whilst we will use equality for the number of authors. Then, we will set weights for each field in order to give more importance to some fields to decide if the articles are referring the same.

9.4.2 Solution

9.4.2.1 Comparison of two articles

In the previous section, we have briefly introduced what we had in mind to compare two articles. Now, it is time to go into detail and specify how we determine if two articles are referring the same or not.

First of all, we divide the fields of the article into three parts depending on what they are referring to:

- Information about the article
- Information about the journal
- Information about the article in the journal

Then, we compare separately each different part and we decide whether parts refer the same article or not. Two articles will be marked as duplicates if all three parts agree that they are referring the same article.

Let's see how we compare the different parts. Even though the result of the comparisons of different parts will be true or false, we internally calculate a grade of similarity (between 0 and 1) and we determine if it is high enough with a given threshold. This grade of similarity is calculated by comparing fields of two articles and pondering the results to give more importance to some fields rather than others. It means we have to assign, for each field, two things: a comparative function and a weight. Table 2 shows this information for the different fields.

For now, we have selected a threshold of 0.6 for all types of field. However, thresholds and weights can be changed in the next iteration, in which we will compare the results obtained from our deduplication process with manually deduplicated data. In terms of weights, they are not random now but comes from manually analysing the blocks resulting from LSH and viewing which fields are more relevant.

Type of field	Field	Comparative Function	Weight
Article	Publication year	Equality	25
	Title	Levenshtein	30
	Number of authors	Equality	10
	Authors	Levenshtein	30
	Article type and Article type description	Or equality[12]	5
Article in Journal	Volume	Equality	10
	Number of Journal	Equality	10
	Initial Page	Equality	20
	End Page	Equality	20
	DOI	Equality	40
Journal	ISSN	Equality	30
	Journal Code (Internal)	Equality	30
	Journal Description (related with Journal Code)	Levenshtein	10
	ISI Code	Equality	30

Table 2 Comparative function fields

[12] At least one of the fields must be equals

9.4.2.2 Implementation

Implementing the process explained before will result in a quirky piece of code most times. Our case will not be an exception, so at this point we want to focus on the cohesion of this comparison with Spark Dataset API rather than the comparison itself.

As we have explained before, using Spark Dataset API allows us to structure our data with *case classes.* Then, *case classes* can encapsulate methods and there is where we have placed the comparative function. By this way, we remain the code outside the *case class* very clear.

On the other hand, the most difficult part of this iteration have been how to transform a Spark Dataset of pairs of article codes (resulting from LSH algorithm) to a Dataset of pairs of articles with all their fields, so that we can compare them pairwise and decide whether they are referring the same article or not. Spark APIs provide different methods to join datasets with the same value for a given column. Nonetheless, the trouble we have had have been about the format in which these functions returns the data. Not because it did not fit our needs but because our code became so dirty after the whole process of transforming the dataset. Eventually, we have found a way to implement this process as clean as possible. The following piece of code shows how we have implemented it:

```
val id1Join = distinctLsh.joinWith(articles, $"id1" === $"code")
                         .toDF("id1", "article1")
val id2Join = distinctLsh.joinWith(articles, $"id2" === $"code")
                         .toDF("id2", "article2")
id1Join.join(id2Join, $"id1" === $"id2").drop("id1", "id2").as[CandidateArticles]
       .filter(candidatePair =>
              candidatePair.article1.isNearDuplicate(candidatePair.article2))
       .map(duplicate => (duplicate.article1.code, duplicate.article2.code))
       .rdd
```

Code 8 Compare pairwise LSH groups

9.4.3 Conclusion

This iteration has not been difficult at all. We have not had any trouble implementing the different parts. However, the major problems have been to find out a way to implement the different parts of this iteration in a clean way. The comparative function described in the previous section will always result in a large amount of code lines, so the only thing we can do here is structure them in the cleanest way and place them in the correct part of the code.

Another thing that makes the code dirty is the structure of some Spark function's result. For example, a Spark function results in a Scala tuple of two more tuples. But the only value which matters to you are the second value of each tuple. Transforming it to a single tuple with the second value of each tuple results in a very ugly piece of code, so we have tried to avoid these situations in as much as possible.

9.5 Iteration 5

9.5.1 Objectives

This is the last iteration of the project, in which we determine if we have done a good job and our program can find most of the duplicated articles in a dataset. In order to do this evaluation, we need to know which are the duplicated articles of the dataset. Otherwise, we could not compare the results from the deduplication process with anything. Thus, we will manually deduplicate the dataset of articles.

After that, we will be able to compare the results obtained from our deduplication process with the manually deduplicated dataset and calculate the Precision and Recall. If we think the results are good enough, our solution will be finalized. However, if we have some false positives or true negatives, we will have to think how we can improve the comparative function.

9.5.2 Solution

9.5.2.1 Manual deduplication

Although the dataset is not so big, it only contains 686 different articles, it would be a lot of work to manually deduplicate the whole dataset without any help. For this reason, we have done this process starting from the output of the LSH algorithm. At this point, we have the articles classified into different blocks in a way that those that have been classified to the same block have similar titles.

By this way, all we have to do is looking at each block and decide whether the articles in that block are referring the same or not. In most cases, we will be able to make this decision only by comparing the different fields of the articles. In some other cases, we will have to search for both articles in the internet and see if they both exist or not.

After this work, we have found six duplicated articles. Most of them have very similar, if not equal, information about the article itself (title, publication year, authors...) and lack of information in some of the other fields. Some others have almost all fields equals. Finally, there are two strange ones in which the information about the article and the article in the journal are practically the same but the journal seems to be very different. Nonetheless, we have seen that both are referring the same article.

9.5.2.2 Automatic deduplication

After executing our deduplication process over the articles dataset, the result has been quite accurate: four articles correctly marked as duplicate and two true negatives. Analysing these true negatives, we see why they have not been marked as duplicates; their Journal information is completely different and with the comparative function we have implemented it is necessary that journal information of two articles is similar in order to be marked as duplicate.

At this point, we can calculate the Precision and Recall to get a value about how good our process is, although it will not be reliable since the amount of duplicated rows in the dataset is only 6.

$$precision = \frac{4}{4} = 1 \quad recall = \frac{4}{6} = 0.66 \quad F = 2 \cdot \frac{4/6}{10/6} = \frac{4}{5} = 0.8$$

The F-measure obtained with the comparative function defined in the previous iteration is quite good. However, as there are only six duplicates in our dataset, we will try to modify our comparative function to see if we can find all the duplicates and obtain a perfect F-measure.

9.5.2.3 Attempts for a perfect F-measure

Given the few amount of duplicated articles in our dataset, we have tried to find a comparative function that could detect all the duplicated articles. At first, it seemed an easy task since both true negatives followed the same pattern: completely different information about the journal but high similarity in all other fields.

After trying several variations of our comparative function, we have given up the search of the perfect F-measure due to the multiple failed attempts. The idea was to; somehow, give more importance to the information about the article in the journal and less to the journal information itself. We have tried to join the comparison of the fields of the article in the journal and the ones from the journal itself into a single comparison, so that the first ones compensate the second ones. We have also tried to take into account if both initial and final page of the article match in the comparison function, since it would rarely be an incidental event. Finally, we have tried to vary the thresholds of the different comparisons and combine them in different ways, for example, marking as duplicates articles with strictly equal information about the article.

Unfortunately, none of the previous attempts increased the F-measure but most of them decreased it dramatically. Moreover, true negatives obtained with the initial comparison function have kept as true negatives with most of the attempts. For this reason, we have decided to keep the initial comparison function. In the end, an F-measure of 0.8 is not a bad value.

9.5.3 Conclusion

Although we have not been able to improve the F-measure of our results, which was our objective for this iteration, we have learned some useful things from the attempt. First of all, when applying data mining techniques, we have to think of a solution which will be applied to a set of most times heterogeneous data. It means we cannot implement a solution adapted to a few number of elements because others will probably be different and it will not work properly. This is what happened to us when we were trying to detect all the duplicates of the dataset. The more we adapted our comparative function to the true negatives, the more false positives we obtained.

In conclusion, when working with big data, everything must be approximate. We cannot expect to have a perfect F-measure because data is most times very heterogeneous and we would rarely find a comparative function producing an F-measure of 1. Moreover, if we eventually found it, it would be because our dataset is not so big. Nonetheless, it could happen that the solution producing a perfect F-measure with the actual dataset produced worse results than the applied comparative function when the amount of data grew up. That would be because we would have adapted so much the solution to the actual dataset.

10 Outcomes

The outcomes of the project have been exposed in the last iteration of it. However, we are going to summarize them here in order to make clear what the results of the project are. Table 3 shows the important rows of the resulting data from the deduplication process. The articles are grouped in pairs so that each pair is a duplicated article. Even though you can see if the deduplication process has detected each row as duplicated or not looking at the field “sameAs”, it is also indicated within different background colors; green if we have detected it and red otherwise. Unfortunately, we could not include all fields of the articles in this table because of the large space it occupied. Nevertheless, we have included the most important ones in order to see the differences between the duplicated articles.

As you can see, our deduplication process has detected four out of six duplicated articles, obtaining an F-measure of 0.8, which is not bad at all. However, as we also have mentioned previously, a dataset with six duplicated articles is not enough to prove our solution, so we will have to wait for a bigger dataset in order to test our solution and see if better than we think or worse.

sameAs	code	year	title	authors	vol.	num.	ini. P	end. P	j. Type	issn	j. Code	journalDesc	isi
	7608	2002	Un modelo ... cultural del area del Montsec	Sendï¿½n M, Granollers T, Lo	16		43	48	Article d'inve	1137-3601	906	Inteligencia Artificial. Revist	
	4585	2002	Un modelo ... cultural del Montsec	Sendï¿½n M, Granollers T, Lo		16	43	48	Article d'inve		200115	AEPIA (Asociacií¿½n Espaï¿	
6698	6698	2004	Coscheduling ... Non-dedicated Linux NOW	Hanzich M, Ginï¿½ F, Solsona	3241	1	327	336	Article d'inve	0302-9743	904459	Lecture Notes in Computer	4459
6698	6865	2004	Coscheduling ... Non-Dedicated Linux Cluster	Hanzich M., Ginï¿½ F, Solson	3241		327	336	Article d'inve	0302-9743	904459	Lecture Notes in Computer	4459
13798	22365	2009	Inquiry based learning for older people at a u	Martorell I, Medrano M, Soli	35	8	712	731	Article docei	0360-1277/	911424	Educational Gerontology	11424
13798	13798	2009	Inquiry based learning for older people at a u	Martorell I, Medrano M, Soli	35	8	712	731	Article d'inve	0360-1277/	911424	Educational Gerontology	11424
	14035	2009	Generating Hard Instances for MaxSAT	Bï¿½jar R, Cabiscol A, Manyï¿			191	195	Article d'inve	0195-623X	201979	Proceedings of the 39th Int	
	14010	2009	Generating Hard Instances for MaxSAT	Ramon Bejar, Alba Cabiscol,			191	195	Article d'inve	0018-9219	901378	Proceedings of the IEEE	1378
21943	21943	2015	Solving the Routing and Wavelength Assignm	Teresa Alsinet									
21943	21977	2015	Solving the Routing and Wavelength Assignm	Alsinet T, Bï¿½jar R, Fernï¿½	28	1	21	34	Article d'inve	0921-7126/	909133	Ai Communications	9133
23461	23461	2016	Non existence of some mixed Moore graphs	Lï¿½pez N, Miret JM, Fernï¿½	339	2	589	596	Article en pr	0012-365X	902061	Discrete Mathematics	2061
23461	24363	2016	Non existence of some mixed Moore graphs	N. Lï¿½pez									

Table 3 Deduplication process output

11 Future work

There are many things we have in mind to do in the future to improve the results or add additional functionalities to the solution. Some of them have been already mentioned in the planning of the project but we could not afford doing them during the established duration of the project. Others came into our minds while developing some features of the project. Next, we are going to describe a bit each one of the future features or improvements.

11.1 Select a duplicate to keep as principal

With the current deduplication process, the only thing we do is adding a new field to each article that indicates, if it is a duplicate article, the code of another row referring the same article. Note that, even though we compare articles pairwise, in the final output all duplicates of the same article refers to the same row. However, this row, which we would call principal row, is selected without any criteria. Therefore, the idea would be to implement some kind of algorithm in order to determine which row contains more valuable information and keep it as principal. In some cases, it could even be good to create a new row with a combination of information from the different duplicated rows to keep it as principal containing the most valuable information.

This is a feature that we had in mind from the beginning of the project. Furthermore, this feature is also part of the process explained in the paper (Zhang, Z., Nuzzolese, A. G., & Gentile, A. L., 2017). Eventually, we have had to leave this feature for future work because of the duration of the project. However, this part will be the first one to be implemented after the finishing of the project.

11.2 Transform data to RDF

In the objectives of the project, we mentioned that we were going to transform the data into RDF in order to continue the deduplication process with that kind of data. This idea came from the paper (Zhang, Z., Nuzzolese, A. G., & Gentile, A. L., 2017), in which they deduplicate linked data. In the beginning, we wanted to follow more or

less the same structure as in the paper to deduplicate our data. However, we have finally gone far from this solution because of different troubles we have encountered during the project, which you may have seen in some of the iterations. With all that, we do not know if we will end up implementing it or not. If we do, it will be to transform the dataset into linked data rather than helping on the deduplication process.

11.3 Generalize the solution

Nowadays, there is a tendency to implement applications as modular as possible. Resulting in applications composed of different modules that can work alone and, consequently, can be reused in many other applications.

In our case, we have not implemented our solution in a modular way. In fact, the whole project depends on the format of the input dataset to be the same we have. Nonetheless, this is something we do not like and we have seen that it would not be difficult to generalize our solution in order to make it modular, which would allow us to use it with different data as long as it fulfils some requirements.

There are two parts of the project that we could extract into separated modules and import them from the project. The first module would contain the LSH algorithm, which is already implemented in a generic way but is still part of the project. The second part would be the deduplication process itself, as we will explain next.

As we have seen during the different iterations of the project, the point where we really deduplicate the articles is when we compare them pairwise. This comparison is done by a comparative function we define inside the case class that represents the structure of the data of our dataset. Before this point, all we need are rows containing an identifier and a string in order to apply LSH. What we would like to do would be to create some kind of abstract class to represent the rows of the dataset only containing an identifier, the string to apply LSH and an abstract comparative function. Then, we would generalize the solution working with this abstract class. By this way, whenever we wanted to apply the deduplication process to a new dataset (with different structure), we would define a case class extending the abstract one and implementing the comparative function.

In our opinion, this improvement would make our solution look better and, since it is open source, maybe someone could be interested in using it. However, we do not really know if this implementation is possible since we are not experts in Scala. According to some research done in this way, it should be possible but we are sure it will not be as simple as we have explained it.

11.4 Publications in books

In this project, we have worked all the time with a dataset of publications in journals from the GREC repository. However, this repository also contains a dataset of the publications in books, which contains some different fields. The idea is to deduplicate this dataset too and even apply the deduplication process to both datasets at the same time. By this way, we will be able to detect if someone has registered an article from a journal as if it were published in a book and vice versa.

Indeed, this feature is very important since we want to eliminate all duplicates in the GREC repository. However, before doing that part, we would better do the previous one (Generalize the solution) as it will ease a lot the task of adapting the algorithm to the fields of the dataset of publications in books.

11.5 Data from all Catalan Universities

There is no doubt the most disappointing part of this project is the fact that only six duplicated rows were present in the test dataset. That dramatically decreases the value and reliability of the project because such amount of duplicated rows is not by far enough to prove the accuracy of our solution.

The fact is that our dataset only contains publications from members of the DIEI (Department of Computer Engineering and Industrial Engineering) of the University of Lleida. However, the GREC repository also contains publications from all other departments of the University, which would be very valuable for our project. In this sense, we contacted with the responsible of the GREC repository and they addressed us to the PRC[13] (Research Portal of Catalan Universities). The PRC is a relatively new initiative whose intention is to display and disseminate from a single

[13] http://portalrecerca.csuc.cat

place all research activity carried out in Catalan universities. It currently contains about 450,000 publications.

Definitely, it would be a very valuable dataset to test our project but, unfortunately, we could not have access to this data yet. Nonetheless, we have already contacted the responsible of the project and they seem to be willing to give us the data. So we hope having this dataset in a near future.

12 Conclusions

Throughout the project, we have had to face different problems of a data science project, which have given us a general idea of what we can expect from these kind of projects. Next, we are going to review all the valuable experience we have gained during the project.

12.1 Data is the most valuable

We have been saying it all the time along the project, the most important thing for a data science project is the data. Note that we are not talking about big data but about data science, since the words “big data” are associated to huge amounts of data. Nonetheless, even to apply data science techniques, which, in fact, are more or less the same we would apply in a big data project, we need a large amount of data in order to try and test our implementation. Data science solutions can also have code tests, which can assert that the code does what you expect, but you will not really know if it works well until you test it with real data and analyses the results.

This fact turns even worse if you need some of the data to accomplish a given pattern. That was our situation, apart from having a dataset, we needed it to contain duplicated rows. Unfortunately, our dataset only contained six duplicated rows, which are the relevant rows in order to test our solution.

In conclusion, before starting a data science project, you must be sure that you have sufficient valuable data to test your solution. Otherwise, you take the risk of ending up your project without knowing whether your solution is accurate or not.

12.2 The solution will never be perfect

When working with large volumes of data, it is not possible to process them all sequentially due to the computational cost it would have. Instead, data science techniques uses some algorithms which compute data faster in exchange of losing accuracy. Our solution cannot be an exception, whilst the most accurate result would be obtained comparing each article with all other articles, the amount of comparisons would be insane. Applying LSH we have avoided most of this

comparisons but we could have two duplicated articles not even being compared, although it would be a rare case.

On the other hand, we have had some trouble trying to find the most accurate comparative function. Our thinking was: given that we only have six duplicated articles, let's try to find them all through our solution. However, data was so heterogeneous that we could not find a comparative function that was specific enough to avoid false positives but general enough to detect all true positives. We can imagine this is not a specific problem of our project but something that happens in most data science projects, so it has been worth to have this trouble so that we obtained more experience.

12.3 Making it work is not the only matter

In data science, like in any other computer science field, for any single problem, there are many different ways of implementing a solution. Some of them involve more code, others less, some of them are more "elegant", others less. However, in data science solutions there is often another factor that makes some solutions "acceptable" and others not: the efficiency.

We have been able to check the importance of the efficiency of an implementation in the second iteration, in which a bad implementation prevented our program to finish unless we worked with few amount of data. Moreover, since we work in Spark and we do not know which functions of the Spark's API are more efficient, sometimes we might not know in which part of the solution the execution is being stuck. In order to see some details about the execution of a Spark's program, it is always good to have access to its User Interface.

This problem can easily happen also in other types of program and we might not even realise it. Nonetheless, in data science a piece of code is usually executed over a large dataset. Consequently, a small efficiency error results in a huge delay in the program execution time.

13 Bibliography

1. Damji, J. (2016, July 14). *A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets.* Retrieved from https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html
2. Jure Leskovec, Anand Rajaraman, Jeff Ullman. (2014). Chapter 3: Finding Similar Items. In A. R. Jure Leskovec, *Mining of Massive Datasets* (pp. 73-130). Cambridge University Press.
3. Odersky, M. (n.d.). *What is Scala?* Retrieved from https://www.scala-lang.org/what-is-scala.html
4. Ullman, J. D. (2015, Sept. 11). Locality-Sensitive Hashing. *Locality Sensitive Hashing Part 1, Jeffry D Ullman*. Retrieved from https://www.youtube.com/watch?v=bQAYY8INBxg
5. Ullman, J. D. (2015, Sept. 11). Locality-Sensitive Hashing. *Locality Sensitive Hashing Part 2, Jeffry D Ullman*. Retrieved from https://www.youtube.com/watch?v=MaqNlNSY4gc
6. Zhang, Z., Nuzzolese, A. G., & Gentile, A. L. (2017). Entity Deduplication on ScholarlyData. *European Semantic Web Conference* (pp. 85-100). Springer. Retrieved from http://www.scholarlydata.org/papers/eswc2017/dataCleaning.html